Optimizacija generiranja dinamičkih mreža kretanja u Unityu

Vinković, Tomislav

Sveučilište Josipa Jurja Strossmayera u Osijeku

Fakultet primijenjene matematike i informatike

Sveučilišni prijediplomski studij Matematika i računarstvo

Optimizacija generiranja dinamičkih mreža kretanja u Unityu

Završni rad

Mentor:

izv. prof. dr. sc. Domagoj Matijević

Komentor:

dr. sc. Luka Borozan

Student:

Tomislav Vinković

Osijek, 2024.

Sadržaj

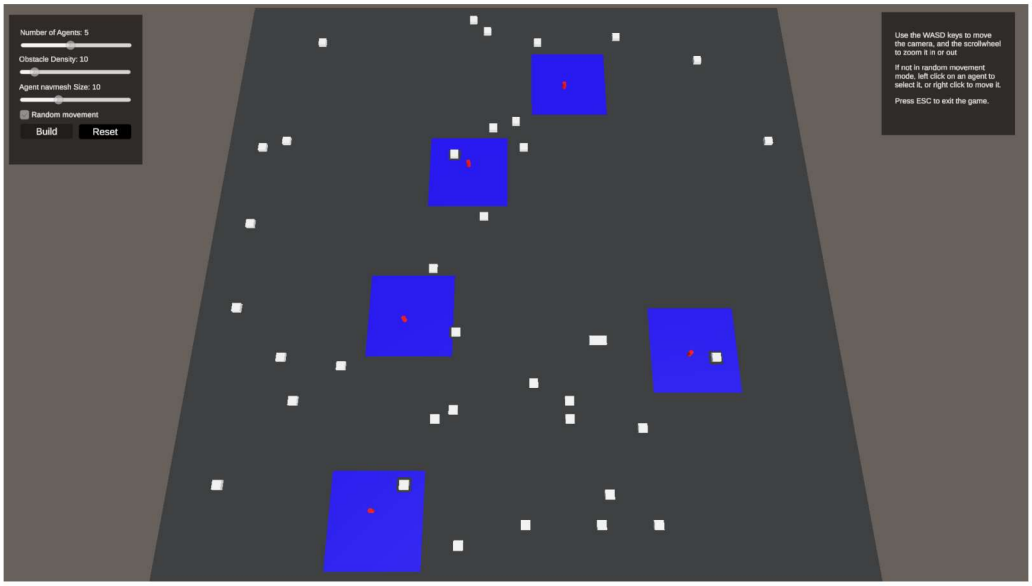
1 | Uvod

Ugrađena implementacija mreža kretanja u Unityu pruža nam sustav za kontrolu AI agenata u igri koji je jednostavan za korištenje. Sve što moramo napraviti kako bismo stvorili mrežu kretanja je instancirati jednu navigacijsku mrežu u sceni i reći Unityu da izradi mrežu kretanja za AI agente. Nakon toga, jedino što trebamo učiniti jest zadati AI agentu ciljnu točku u sceni, i on će se samostalno kretati do te točke.

Međutim, izrada mreže kretanja za cijelu scenu unaprijed nije uvijek moguća. Najbolji primjer toga su igre sa proceduralnim generiranjem terena. U tom slučaju, jedna od popularnih metoda izrade mreža kretanja je instanciranje lokalne, dinamičke mreže kretanja oko svakog agenta, koja se ažurira dok se agent kreće po sceni. Veliki problem ove metode je brzina. Naime, u naivnoj implementaciji mreža se ponovno generira oko agenta kad god se on pomakne. Također, postoji i problem preklapanja dvaju mreža. Dvije navigacijske mreže ne smiju se preklapati jer tada agenti mogu imati poteškoća s pronalaskom puta do zadane točke.

Optimizacija predstavljena u ovom završnom radu rješava ovaj problem. Korištenjem ove optimizacije, navigacijske mreže neće se ponovno generirati kad god se agenti pomaknu, već će se generirati kad se jedan od agenata unutar mreže pomakne dovoljno blizu rubu navigacijske mreže. Problem preklapanja navigacijskih mreža riješen je jednostavnim algoritmom klasteriranja po dvije osi.

Optimizacija je implementirana u sklopu Unity projekta, koji osim samog algoritma sadrži sve značajke za vizualizaciju agenata i navigacijskih mreža. Također, aplikacija sadrži pokretnu kameru sa mogućnostima kretanja u svim smjerovima i promjene razine uvećanja, kao i korisničko sučelje za promjene postavki simulacije. Konačna aplikacija vidi se na slici 1.1.



Slika 1.1: Konačna aplikacija

2 | Struktura Unity projekta

Prije implementacije, moramo kreirati novi Unity projekt. Unity ima nekoliko predložaka koji nam mogu pomoći brže razviti vrstu projekta koji želimo. S obzirom na to da želimo koristiti trodimenzionalno okruženje, započet ćemo novi projekt koristeći 3D predložak.

Unity projekt sačinjen je od nekoliko važnih komponenti:

- Skripte - ovdje se nalazi sav naš programski kod
- Prefabi - ovdje se nalaze svi predlošci za objekte u igri, poput agenta i navigacijske mreže
- Scene - sve scene unutar projekta. Naš projekt ima jednu scenu
- Materijali, animacije, audio i ostali resursi - sve audiovizualne datoteke aplikacije

Programski kod aplikacije podijeljen je u sljedeće cjeline:

- Kontroleri - služe kontroli toka i mijenjanju stanja u igri
- Generatori - služe generiranju nivoa, agenata i vidljivih obrisa navigacijskih mreža u igri
- *Math* - potrebni matematički konstrukti i algoritmi
- *Mesh* - služi generiranju navigacijskih mreža
- *World* - skup jasno definiranih konstantnih vrijednosti i statičkih metoda vezanih za scenu
- *Enums* - kup jasno definiranih konstantnih vrijednosti koje omogućuju lako upravljanje stanjima
- *Camera* - kontrola kamere u igri

3 | Postavljanje scene

Kako bismo mogli kreirati navigacijske mreže i omogućiti agentima da se kreću po njima, moramo postaviti našu scenu u Unity-u. Želimo da korisnik aplikacije ima kontrolu nad nekim postavkama simulacije, pa ćemo scenu postavljati dinamički, kroz kod. Za dinamično postavljanje scene kroz kod koristit ćemo prefabe. Prefabi u Unity-u nam omogućuju da objekt sa identičnim svojstvima instanciramo više puta. Mi ćemo u projektu koristiti nekoliko prefaba:

- *Agent* - agent koji se kreće po navigacijskoj mreži
- *Ground* - površina po kojoj se agenti kreću. Predstavljat će ju kvadar proizvoljnih dimenzija
- *Wall* - zidovi koji predstavljaju prepreke agentima. Predstaljat će ih kvadar konstantnih dimenzija.
- *DynamicNavMesh* - objekt koji sadržava naš kontroler za dinamičku mrežu kretanja i *Nav Mesh Surface* komponentu

3.1 Agent

Naš agent bit će predstavljen uspravnim cilindrom. On već ima ugrađene komponente *MeshRenderer* (za prikaz cilindra) i *CapsuleCollider* (kako bi Unity znao dodiruje li se cilindar s nekim drugim objektom u sceni). Dodat ćemo mu *RigidBody* komponentu kako bi bio pod utjecajem fizike u igri. To ćemo učiniti pritiskom *Add component* dugmeta u *Inspector* prozoru, te pritiskom na *RigidBody* komponentu u brzoj pretrazi. Također, dodat ćemo mu i novu, praznu skriptu koju ćemo nazvati *AgentMovement*. Ovu skriptu ćemo kasnije iskoristiti za programiranje kretnji agenta.

3.2 Korisničko sučelje

Korisničko sučelje aplikacije izrađeno je pomoću ugrađenih Unity komponenti za izgradnju korisničkih sučelja. Sučelje se sastoji od postavki za promjenu simulacije s lijeve strane i uputa za korištenje s desne strane. Sučelje je vidljivo na slici 3.1.

Slika 3.1: Korisničko sučelje

Postavke simulacije s lijeve strane sadrže sljedeće elemente:

- Pomični element za postavljanje broja agenata
- Pomični element za postavljanje gustoće prepreka
- Pomični element za postavljanje veličine navigacijske mreže oko jednog agenta
- Prekidač za uključivanje/isključivanje nasumičnog kretanja agenata. Ako je ovaj prekidač isključen, agenti se neće kretati sami od sebe, već ćemo ih mi moći ručno pomicati
- Gumb *Build*, koji služi za izgradnju scene sa zadanim postavkama
- Gumb *Reset*, koji uništava trenutnu scenu (ukoliko postoji) i omogućuje nam promjenu postavki i ponovnu izgradnju scene

Za kontrolu korisničkog sučelja zadužena je klasa *GameStateController*. Ova klasa ima reference na sve elemente korisničkog sučelja u igri i reagira na promjene stanja u korisničkom sučelju. Reference na elemente sučelja postavljamo u *Start* metodi. Elementima sučelja možemo pridružiti metode za različite radnje nad tim elementima (promjena vrijednosti, klik) pomoću metode *AddListener*.

```
1 void Start()
2     {
3         ...
4         agentSlider.onValueChanged.AddListener(delegate {
    UpdateNumberOfAgents(); });});
5         ...
6     }
```

3.3 Izgradnja nivoa

Za izgradnju nivoa s preprekama i agentima zadužena je klasa *WorldBuilderController*. Klikom na gumb *Build* u korisničkom sučelju pokrećemo metodu *HandleBuildButtonPressed* u klasi *GameStateController*, koja pokreće javnu metodu *Build* iz klase *WorldBuilderController*.

```
public void Build(int numberOfAgents, float obstacleDensity)
{
    if(State == WorldBuilderState.Standby)
    {
        State = WorldBuilderState.Building;
        StartCoroutine(BuildWorld(numberOfAgents, obstacleDensity))
    ;
    }
}
```

Metoda *Build* pokreće metodu *BuildWorld*, koja je zadužena za izgradnju nivoa.

```
IEnumerator BuildWorld(int numberOfAgents, float obstacleDensity)
{
    agentGeneratorController.PlaceAgents(numberOfAgents);
    levelGeneratorController.GenerateLevel(obstacleDensity);

    State = WorldBuilderState.Ready;

    globalNavMeshController.MarkForUpdate();

    yield return null;
}
```

Metoda započinje postavljanjem agenata na nasumična mjesta u sceni pomoću metode *PlaceAgents* iz klase *AgentGenerator*. Zatim, postavlja zidove (prepreke za agente) koristeći metodu *GenerateLevel* iz klase *LevelGenerator*. Detaljnije ćemo proći ove dvije metode u sljedeće 2 sekcije.

Metoda *BuildWorld* odvija se unutar Unity korutine. Korutine omogućuju da zahtjevne i spore operacije, poput izgradnje nivoa, podijelimo na više sličica. Na taj način, kod postaje asinkron i neblokirajući, te igrač može nastaviti upravljati igrom dok se spora operacija odvija u pozadini. Na kraju svake korutine potrebno je izvršiti naredbu *yield*, nakon koje slijedi naredba *return*.

```
yield return null;
```

Korutina može vratiti *null*, što znači da se korutina nastavlja odmah na sljedećoj sličici. Također, može vratiti instancu klase *WaitForSeconds*, koja određuje fiksno čekanje izraženo u sekundama nakon kojeg će korutina nastaviti s radom.

Važno je naglasiti da korutine nisu zamjena za pravi multithreading. Međutim,

one predstavljaju jednostavniju alternativu koja ne zahtijeva korištenje muteksa i zaključavanja resursa između niti.

3.4 Postavljanje agenata u sceni

Kao što smo napisali u prethodnoj sekciji, postavljanje agenata u sceni odvija se u metodi *PlaceAgents* iz klase *AgentGenerator*.

```
public void PlaceAgents(int numberOfAgents)
{
    for (int i = 0; i < numberOfAgents; i++)
    {
        Vector3 pos = new Vector3(
            Random.Range(-gameStateController.PlaneWidth / 2f,
    gameStateController.PlaneWidth / 2f), 1.5f,
            Random.Range(-gameStateController.PlaneHeight / 2f,
    gameStateController.PlaneHeight / 2f)
        );
        Instantiate(agent, pos, Quaternion.identity, transform);
    }
    AreAgentsGenerated = true;
}
```

Metoda u obzir uzima visinu i širinu površine po kojoj će se agenti kretati. Zatim, u odnosu na broj agenata koji je korisnik odabrao u korisničkom sučelju, metoda će postaviti toliko agenata na nasumične pozicije u sceni. Metoda instancira agente naredbom *Instantiate* iz Unity-a.

3.5 Postavljanje prepreka u sceni

Kao što smo napisali u prethodnoj sekciji, postavljanje prepreka u sceni odvija se u metodi *GenerateLevel* iz klase *LevelGenerator*. Metoda u obzir uzima visinu i širinu površine po kojoj će se agenti kretati, i na temelju toga pokreće 2 *for* petlje, po visini i širini

```
for (int x = 0; x <= WorldWidth; x+=(int)wallWidth)
    {
        for (int z = 0; z <= WorldHeight; z+=(int)wallWidth)
        {
            ...
```

Prije postavljanja zida, metoda mora provjeriti presijeca li taj zid nekog agenta, i presijeca li taj zid jednu od ciljnih točaka koju agenti koriste prilikom nasumičnog kretanja. Provjera presijecanja radi se pomoću *Bounding box* algoritma.

```
if (UnityEngine.Random.value > 1 - obstacleDensity)
{
```

```
3       BoundingBoxXZ wallBounds = new BoundingBoxXZ(
4           x-wallWidth/2,
5           x+wallWidth/2,
6           z-wallWidth/2,
7           z+wallWidth/2
8       );
9       bool intersectsWaypoint = DoesWallIntersectWaypoint(wallBounds)
        ;
10      if(intersectsWaypoint) continue;
11
12      bool intersectsAgent = DoesWallIntersectAgent(wallBounds);
13      if(intersectsAgent) continue;
14
15
16      // Spawn a wall
17      Vector3 pos = new Vector3(
18          x - WorldWidth / 2f,
19          1.5f,
20          z - WorldHeight / 2f
21      );
22      Instantiate(wall, pos, Quaternion.identity, transform);
23  }
```

Prije obavljanja provjera za presijecanje, bitno je vidjeti hoće li zid uopće potencijalno biti na mjestu gdje želimo obaviti provjere. Postavljanje zidova je nasumično, a vjerojatnost hoće li zid biti postavljen na neku lokaciju ovisi o odabiru korisnika u korisničkom sučelju. Metode *DoesWallIntersectAgent* i *DoesWallIntersectWaypoint* su interne metode klase.

Pošto je ciljnih točaka malo, metoda *DoesWallIntersectWaypoint* jednostavno za svaku točku provjerava presijeca li ju potencijalni zid. Sve ciljne točke spremljene su u klasi *AgentManager*.

```
1   bool DoesWallIntersectWaypoint(BoundingBoxXZ wallBounds)
2   {
3       // Get the waypoints from the agent manager
4       List<Vector3> waypoints = agentManager.AgentWaypoints;
5
6       // Check if the wall intersects with any of the waypoints
7       foreach (Vector3 waypoint in waypoints)
8       {
9           if (wallBounds.Intersects(waypoint))
10          {
11              return true;
12          }
13      }
14
15      return false;
16  }
```

Metoda *DoesWallIntersectAgent* provjerava presijeca li zid nekog agenta.

```
private bool DoesWallIntersectAgent(BoundingBoxXZ wallBounds)
{

    float agentWidth = World.AGENT_WIDTH;

    float minX = wallBounds.minX;
    float maxX = wallBounds.maxX;

    int startIndex = FindFirstAgentInRange(minX);
    if (startIndex == -1) return false; // No agents in range

    for (int i = startIndex; i < agents.Count; i++)
    {
        Vector3 agentPosition = agents[i].transform.position;
        if (agentPosition.x > maxX) break;
        if (agentPosition.z >= wallBounds.minZ && agentPosition.z
    <= wallBounds.maxZ)
        {
            BoundingBoxXZ agentBounds = new BoundingBoxXZ(
                agentPosition.x - agentWidth / 2,
                agentPosition.x + agentWidth / 2,
                agentPosition.z - agentWidth / 2,
                agentPosition.z + agentWidth / 2
            );

            if (wallBounds.Intersects(agentBounds))
            {
                return true;
            }
        }
    }
    return false;
}
```

Metoda koristi binarno pretraživanje na listi agenata postavljenoj unutar metode *GenerateLevel*.

```
public void GenerateLevel(float obstacleDensity)
{
agents = World.GetActiveAgents();

// Sort agents by their x and z positions
agents.Sort((a, b) =>
{
    int compareX = a.transform.position.x.CompareTo(b.transform.
    position.x);
    return compareX != 0 ? compareX : a.transform.position.z.
    CompareTo(b.transform.position.z);
});
...
```

Agenti su sortirani po x-osi, što znači da će binarno pretraživanje pronaći prvog

agenta čija x koordinata upada u raspon

$$< xmin, xmax >$$

Zatim, metoda od tog agenta, pa do prvog koji izlazi iz raspona po x-osi redom provjerava presijeca li *bounding box* tog agenta i po z-osi. Ako presijeca bar jednog agenta i po x i po z-osi, to znači da ne smijemo ovdje postaviti zid, pa vraćamo true (jer *bounding box* presijeca barem jednog agenta).

4 | Generiranje dinamičkih mreža kretanja

Generiranje dinamičkih mreža kretanja u našoj aplikaciji sastoji se od nekoliko koraka. Generiranje započinje klasteriranjem agenata u sceni. Za svaki klaster ćemo generirati jednu navigacijsku mrežu. Nakon klasteriranja, generiraju se navigacijske mreže koje se spremaju u red *updateQueue GlobalNavMeshController*-a (mreže se prvo samo generiraju kao prazni objekti u sceni, nisu još izgrađene). Mreže se potom grade u *GlobalNavMeshController*-u unutar korutine.

4.1 Početak generiranja

Kontrola životnog ciklusa svih navigacijskih mreža u sceni odvija se u klasi *GlobalNavMeshController*. U ovoj klasi ćemo prvo definirati *Start* i *Update* metode.

```
...

void Start()
{
    // mark for first update
    MarkForUpdate();

    // start the update queue
    StartCoroutine(ProcessUpdateQueue());
}

void Update() {
    if(ShouldUpdate && worldBuilderController.State ==
    WorldBuilderState.Ready) {
        RecalculateNavMeshes();
    }
}
```

Start metoda započinje pozivanjem metode *MarkForUpdate*, koja signalizira globalnom kontroleru da je vrijeme za prvo ažuriranje navigacijskih mreža. Na kraju, metoda pokreće korutinu *ProcessUpdateQueue*, koju ćemo detaljnije obraditi u narednoj sekciji.
Update metoda, koja se ponavlja svaku sličicu, provjerava je li potrebno ažurirati neke od navigacijskih mreža. Navigacijske mreže bi se trebale ažurirati ako i samo

ako je *GlobalNavMeshController* označen za ažuriranje i ako je simulacija gotova s procesom izgradnje.

4.2 Životni ciklus navigacijske mreže

Kroz svoj životni ciklus, navigacijska mreža promijeni 5 stanja

- *Build* - navigacijska mreža je spremna za izgradnju
- *Building* - navigacijska mreža je u procesu izgradnje
- *Ready* - navigacijska mreža je u upotrebi
- *Destroy* - navigacjska mreža je spremna za uništenje
- *Destroying* - navigacijska mreža je u procesu uništenja

Svakoj je navigacijskoj mreži pri instanciranju dodijeljeno stanje *Build*.

Za konstantnu i asinkronu izgradnju navigacijskih mreža odgovorne su dvije metode: *RecalculateNavMeshes* i *ProcessUpdateQueue*.
Metoda *RecalculateNavMeshes* odgovorna je za

- Pripremu novih navigacijskih mreža
- Deaktivaciju agenata dok se njihove navigacijske mreže grade
- Popunjavanje reda *updateQueue* sa novim navigacijskim mrežama koje je potrebno izgraditi ili uništiti

Metoda prvo deaktivira sve agente u sceni (jer agent bi smio biti aktivan ako nije postavljen na navigacijsku mrežu). Metoda zatim klasterira agente metodom *ClusterAgents* iz klase *AgentClustering*. Više o algoritmu klasteriranja pričat ćemo u poglavlju 5.

```
1  void RecalculateNavMeshes() {
2      var agents = World.GetActiveAgents();
3      foreach( var agent in agents ) {
4          agent.GetComponent<NavMeshAgent>().enabled = false;
5      }
6      // cluster agents
7      var agentClusters = AgentClustering.ClusterAgents();
8
9      foreach( var (_, surfaceController) in navMeshSurfaces ) {
10         surfaceController.State = DynamicNavMeshState.Destroy;
11         updateQueue.Enqueue(surfaceController);
12     }
13     ...
14 }
```

Na kraju, metodom *BuildNavMeshesFromAgentClusters* iz klase *NavMeshBuilder*, metoda gradi navigacijske mreže na temelju dobivenih klastera.

```
    ...
    // from agent clusters, create navmesh surfaces
    navMeshSurfaces = navMeshBuilder.
    BuildNavMeshesFromAgentClusters(agentClusters);
    foreach( var (_, surfaceController) in navMeshSurfaces ) {
        // assign the global navmesh controller
        surfaceController.GlobalNavMeshController = this;
        updateQueue.Enqueue(surfaceController);
    }
}
```

Metoda *ProcessUpdateQueue* odgovorna je za izdvajanje prve sljedeće navigacijske mreže iz reda *updateQueue* i izvršavanje odgovarajuće operacije nad tom mrežom.

```
IEnumerator ProcessUpdateQueue()
{
    while (true)
    {
        if (updateQueue.Count > 0)
        {
            var surfaceController = updateQueue.Dequeue();
```

Ukoliko je mreži dodijeljeno stanje *Build*, metoda će joj dodijeliti stanje *Building* i započeti proces izgradnje mreže

```
if(surfaceController.State == DynamicNavMeshState.Build)
{
    surfaceController.State = DynamicNavMeshState.Building;
    surfaceController.BuildNavMesh();
    ...
}
```

Ukoliko je mreži pak dodijeljeno stanje *Destroy*, metoda će joj dodijeliti stanje *Destroying* i započeti proces uništenja mreže

```
else if(surfaceController.State == DynamicNavMeshState.Destroy)
{
    surfaceController.State = DynamicNavMeshState.Destroying;
    Destroy(surfaceController.gameObject);
    ...
}
```

4.3 DynamicNavMeshController

Klasa *DynamicNavMeshController* sadrži logiku potrebnu za upravljanje navigacijskom mrežom. Svaka navigacijska mreža u aplikaciji ima svoj pripadni kontroler. Glavne funkcionalnosti klase *DynamicNavMeshController* su:

- Konačna izgradnja mreže u dimenzijama određenim *NavMeshBuilder* objektom
- Praćenje agenata unutar mreže
- Aktivacija agenata nakon izgradnje mreže

Za konačnu izgradnju mreže odgovorna je metoda BuildNavMesh. Metoda započinje zadavanjem centra i dimenzija *navMeshSurface* komponenti (instanca Unity klase *NavMeshSurface*). Pošto se centar *navMeshSurface* komponente gleda u lokalnim koordinatama, moramo postaviti *navMeshSurface.center* na nul-vektor. Dimenzije su prethodno određene u *NavMeshBuilder* objektu.

```
1 public void BuildNavMesh()
2 {
3     navMeshSurface.collectObjects = CollectObjects.Volume;
4     navMeshSurface.center = Vector3.zero;
5     navMeshSurface.size = new Vector3(
6         navMeshBounds.size.x,
7         1f,
8         navMeshBounds.size.z
9     );
```

Nakon određivanja centra i dimenzija, potrebno je samo pozvati metodu *BuildNavMesh* na *navMeshSurface* komponenti kako bi se navigacijska mreža izgradila. Pošto je navigacijska mreža izgrađena, spremna je za korištenje, pa joj dodjeljujemo stanje *Ready*.

```
1 navMeshSurface.BuildNavMesh();
2 State = DynamicNavMeshState.Ready;
```

Na kraju je potrebno ažurirati listu lokalnih agenata koje moramo pratiti, te aktivirati te agente kako bi se mogli početi kretati po mreži. Agente koji su unutar mreže provjeravamo tako da za svakog agenta u sceni provjerimo sadržavaju li dimenzije mreže njegovu poziciju.

```
1 foreach (var agent in agents)
2 {
3     if (navMeshBounds.Contains(agent.transform.position - Vector3.
    up))
4     {
5         agentsInside.Add(agent);
6     }
```

```
7 }
8 agentsInside.ForEach(agent => ReactivateAgentIfOnNavMesh(agent));
```

Agente unutar mreže pratimo u *Update* metodi životnog ciklusa. Više o praćenju agenata pričat ćemo u poglavlju 6.

4.4 NavMeshBuilder

Klasa *NavMeshBuilder* sadrži logiku potrebnu za izgradnju navigacijske mreže. Unutar nje nalaze se 2 metode:

- *BuildNavMeshesFromAgentClusters*
- *InstantiateDynamicNavMeshSurface*

Metoda *BuildNavMeshesFromAgentClusters* prima rječnik klastera agenata u sceni, a vraća rječnik *DynamicNavMeshController* objekata. Potrebu za rječnicima kod klasteriranja ćemo detaljno analizirati u poglavlju 5.
Metoda svakom klasteru u riječniku pridružuje odgovarajuću navigacijsku mrežu. Prvo određuje dimenzije nove mreže pozivanjem metode GetBoundingBox

```
1 public Dictionary<(int, int), DynamicNavMeshController>
    BuildNavMeshesFromAgentClusters(
2         Dictionary<(int, int), AgentCluster> agentClusters
3 )
4 {
5     var navMeshSurfaces = new Dictionary<(int, int),
    DynamicNavMeshController>();
6
7     foreach (var (key, cluster) in agentClusters)
8     {
9         BoundingBoxXZ boundingBox = cluster.GetBoundingBoxXZ();
```

Metoda *GetBoundingBoxXZ* klase *AgentCluster* određuje veličinu nove navigacijske mreže tako da odredi 4 vrha pravokutnika. Vrhovi pravokutnika određeni su najmanjim, odnosno najvećim x i z pozicijama agenata. Zatim obje stranice tog pravokutnika povećava za neku unaprijed određenu konstantnu vrijednost (kako ne bismo odmah imali agente na rubovima mreže, što bi značilo potrebu za ponovnim generiranjem te mreže).

Nakon određivanja dimenzija mreže, instancirati ćemo novu instancu *NavMeshSurface* prefaba korištenjem *InstantiateDynamicNavMeshSurface* metode.

```
1 var navMeshSurface = InstantiateDynamicNavMeshSurface(boundingBox.
    center);
```

Metoda *InstantiateNavMeshSurface* je jednostavan *wrapper* oko standardne *Instantiate* metode u Unityu, koja služi instanciranju novog objekta korištenjem prefaba koristeći podatke o proporcijama mreže i njenom centru.
Na kraju petlje, metoda *BuildNavMeshesFromAgentClusters* će u rječnik spremiti kontroler novoinstanciranog *DynamicNavMeshSurface* objekta. Metoda vraća rječnik *navMeshSurfaces*.

```
1         var surfaceController = navMeshSurface.GetComponent<
    DynamicNavMeshController>();
```

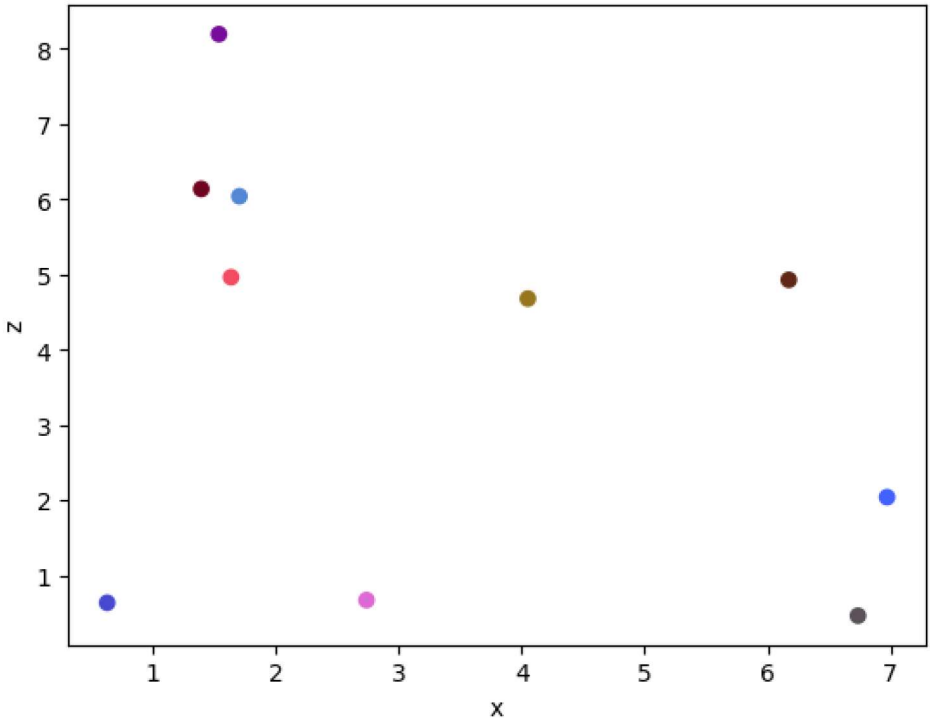
```
2
3           surfaceController.SetNavMeshBounds(
4               new Bounds(boundingBox.center, boundingBox.size)
5           );
6           navMeshSurfaces[key] = surfaceController;
7       }
8
9       return navMeshSurfaces;
10  }
```

5 | Klasteriranje agenata

Cilj klasteriranja u našoj aplikaciji je grupirati agente na temelju njihove međusobne udaljenosti. Algoritam treba moći obraditi veliki broj agenata u kratkom vremenu, kako bi kod iz prethodno napisanih poglavlja što prije mogao početi s izgradnjom novih mreža.

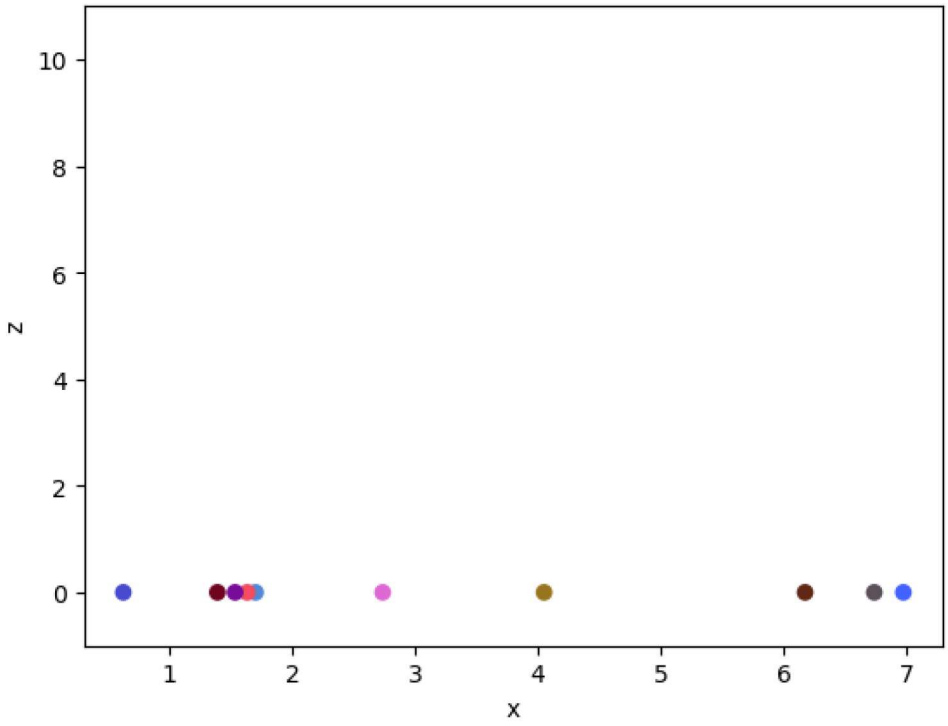
5.1 Ideja algoritma

Raspored agenata u sceni po xz ravnini prikazat ćemo ravninom omeđenom osima x i z. Svaki agent predstavljen je jednom točkom na toj ravnini. Jedan primjer rasporeda agenata vidljiv je na slici 5.1.



Slika 5.1: Agenti na xz ravnini

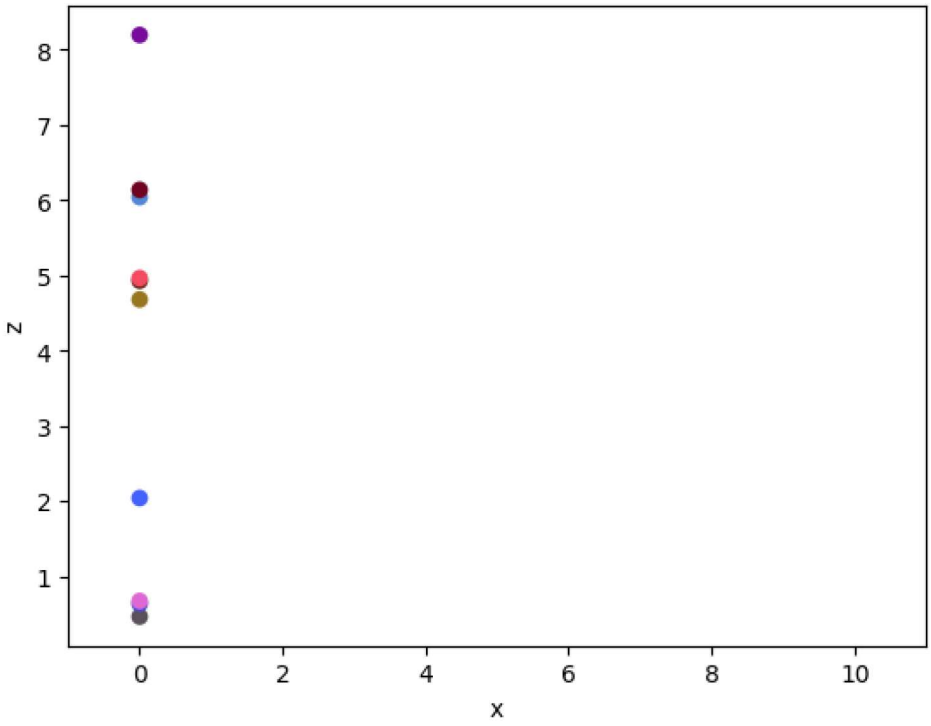
Projicirajmo sve agente na x os.



Slika 5.2: Projekcija agenata na x os

Primjetimo da je proces klasteriranja puno lakši ukoliko agente klasteriramo po samo jednoj osi. Dovoljno je uzeti proizvoljnu duljinu d. Započet ćemo tako da ćemo agenta s najmanjom vrijednošću x staviti u prvi klaster. Za svakog sljedećeg agenta, gledamo je li njegova udaljenost do zadnjeg agenta u trenutnom klasteru manja od d d. Ukoliko je, ta dva agenta su na x osi unutar istog klastera. Ukoliko nije, započinjemo novi klaster s tim agentom kao prvim agentom u klasteru.

Isto možemo napraviti i na z osi. Za duljinu jednog klastera uzimamo jednaku duljinu d koju smo uzeli kod klasteriranja po x osi.



Slika 5.3: Projekcija agenata na z os

Nakon što smo napravili klasteriranje na obje osi, imat ćemo k klastera na osi x i l klastera na osi z. Dvodimenzionalno klasteriranje ćemo napraviti tako da

- Sve agente koji se nalaze u klasteru 1 na x osi i istovremeno u klasteru 1 na z osi smještamo u novi klaster koji označavamo s $\{1, 1\}$.
- Sve agente koji se nalaze u klasteru 1 na x osi i istovremeno u klasteru 2 na z osi smještamo u novi klaster koji označavamo s $\{1, 2\}$.
  ...
- Sve agente koji se nalaze u klasteru k na x osi i istovremeno u klasteru l na z osi smještamo u novi klaster koji označavamo s $\{k, l\}$.

5.2 Implementacija algoritma klasteriranja

Implementacija algoritma klasteriranja u našoj aplikaciji nalazi se u klasi *AgentClustering*. Unutar nje se nalaze 3 metode

- *ClusterAgents*
- *GroupAgents*
- *CombineClusters*

Metoda *ClusterAgents* započinje dohvaćanjem svih dostupnih agenata u sceni. Zatim, radi projekcije na osi. Projekciju u implementaciji radimo sortiranjem agenata po x, odnosno z koordinati.

```
1 static List<List<GameObject>> GroupAgents(List<GameObject> agents,
    Vector3 direction) {
2     var agents = World.GetActiveAgents();
3     var agentsX = agents.OrderBy(agent => agent.transform.position.
    x).ToList();
4     var agentsZ = agents.OrderBy(agent => agent.transform.position.
    z).ToList();
5 ...
```

Ugrađena implementacija sortiranja liste u C# programskom jeziku za primitivne tipove podataka (poput *int*, *float*, *double*) je implementacija *Intro Sort* algoritma, koji je kombinacija 3 algoritma za sortiranje:

- *Quick Sort*, koji se koristi za većinu sortiranja
- *Heap Sort*, koji se zamjenjuje *Quick Sort* ukoliko dubina rekurzije *Quick Sort* algoritma postaje prevelika (čime se sprječava najgora moguća vremenska složenost $O(n^2)$)
- *Insertion sort*, koji se koristi za male podnizove i nizove s malo elemenata. Iako ima najgoru moguću vremensku složenost $O(n^2)$, zapravo je efikasniji za nizove koji su gotovo sortirani, te za male podnizove zbog korištenja rekurzije u *Quick Sort* algoritmu.

Ovaj algoritam ima prosječnu vremensku složenost $O(n * log(n))$ i prikladan je za našu optimizaciju.

Metoda zatim klasterira agente po osima koristeći metodu *GroupAgents*.

```
1 ...
2 // group agents by x axis
3 List<List<GameObject>> agentClustersX = GroupAgents(agentsX,
    LinearAlgebra.XAxis);
4 // group agents by z axis
5 List<List<GameObject>> agentClustersZ = GroupAgents(agentsZ,
    LinearAlgebra.ZAxis);
6 ...
```

Metoda *GroupAgents* klasterira agente po jednoj osi na način koji smo definirali u poglavlju 5.1. Metoda prvo definira novu listu u koju ćemo spremati klastere i referencu na trenutni klaster.

```
1 ...
2 List<List<GameObject>> agentClusters = new List<List<GameObject>>()
    ;
3 List<GameObject> currentCluster = new List<GameObject>();
4 ...
```

Zatim, za svakog agenta, metoda provjerava nalazi li se u trenutnom klasteru, osim specijalno za prvog agenta, koji je dodijeljen prvom klasteru. Ukoliko je

agent unutar granica trenutnog klastera, metoda ga dodaje listi agenata u klasteru. U suprotnom, kreiran je novi klaster s trenutnim agentom kao prvim agentom novog klastera.

```
float d = 2*gameStateController.AgentNavmeshSize;
for (int i = 0; i < agents.Count; i++) {
    if (i == 0) {
        currentCluster.Add(agents[i]);
        continue;
    }

    bool isInCluster = false;

    // Check by the specified axis
    if (direction == LinearAlgebra.XAxis) {
        float distanceX = Mathf.Abs(agents[i].transform.position.x
   - currentCluster.Last().transform.position.x);
        if (distanceX < d) {
            isInCluster = true;
        }
    } else if (direction == LinearAlgebra.ZAxis) {
        float distanceZ = Mathf.Abs(agents[i].transform.position.z
   - currentCluster.Last().transform.position.z);
        if (distanceZ < d) {
            isInCluster = true;
        }
    }

    if (isInCluster) {
        currentCluster.Add(agents[i]);
    } else {
        agentClusters.Add(currentCluster);
        currentCluster = new AgentCluster { agents[i] };
    }
}
...
```

Na kraju, van petlje, metoda dodaje posljednji klaster u listu (pošto nikad nije dodan unutar petlje) i vraća listu svih klastera.

```
agentClusters.Add(currentCluster);
return agentClusters;
...
```

Vratimo se na metodu *ClusterAgents*. Nakon klasteriranja agenata po osima, metoda prosljeđuje oba klastera metodi *CombineClusters* i vraća rezultat njenog izvršavanja.

```
return CombineClusters(agentClustersX, agentClustersZ);
```

Metoda *CombineClusters* prima dvije liste klastera koje smo dobili klasteriranjem po osima i vraća novi rječnik klastera. Ključ rječnika je uređeni par cijelih brojeva, a vrijednost je lista klastera. Ukoliko se referenciramo na poglavlje 5.1, vidjet ćemo da ovakav konstrukt odgovara konačnom koraku algoritma. Metoda započinje instanciranjem praznog rječnika klastera i instanciranjem dva rječnika koja za ključeve imaju reference na agente, a kao vrijednosti imaju cijele brojeve

```
1  static Dictionary<(int, int), List<GameObject>> CombineClusters
2  (
3      List<List<GameObject>> agentClustersX,
4      List<List<GameObject>> agentClustersZ
5  )
6  {
7      // Create a dictionary to store the combined clusters
8      Dictionary<(int, int), List<GameObject>> combinedClusters = new
       Dictionary<(int, int), List<GameObject>>();
9
10     // Map each GameObject to its cluster index in the x axis
11     Dictionary<GameObject, int> xClusterMap = new Dictionary<
       GameObject, int>();
12     for (int i = 0; i < agentClustersX.Count; i++)
13     {
14         foreach (var agent in agentClustersX[i])
15         {
16             xClusterMap[agent] = i;
17         }
18     }
19
20     // Map each GameObject to its cluster index in the z axis
21     Dictionary<GameObject, int> zClusterMap = new Dictionary<
       GameObject, int>();
22     for (int i = 0; i < agentClustersZ.Count; i++)
23     {
24         foreach (var agent in agentClustersZ[i])
25         {
26             zClusterMap[agent] = i;
27         }
28     }
29     ...
```

Ovaj neobičan konstrukt ima svoju svrhu. Naime, omogućuje nam grupiranje klastera tako što olakšava dohvaćanje indeksa klastera na obje osi za svakog agenta. Sve što trebamo učiniti je referencirati oba rječnika za svakog agenta (jedan rječnik za svaku od osi).

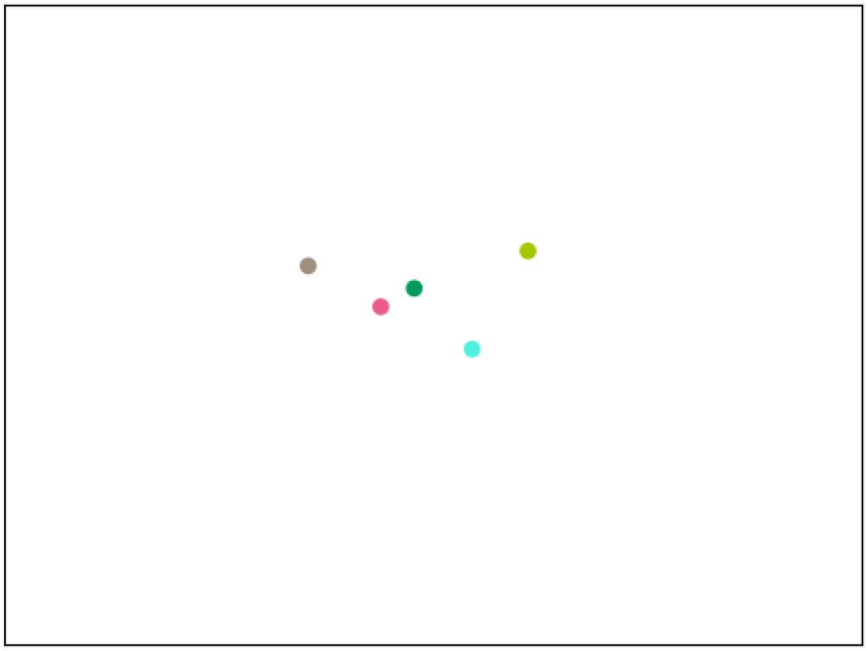
```
1  ...
2  foreach (var agent in xClusterMap.Keys)
3  {
4      int xCluster = xClusterMap[agent];
5      int zCluster = zClusterMap[agent];
```

Na taj način možemo lako odrediti u kojem se klasteru agent nalazi po x osi i u kojem se klasteru nalazi po z osi. Zatim ćemo agenta smjestiti u odgovarajući klaster u *combinedClusters* rječniku. Nakon klasteriranja, metoda vraća rječnik klastera po obje osi.

```
    var combinedKey = (xCluster, zCluster);
    if (!combinedClusters.ContainsKey(combinedKey))
    {
        combinedClusters[combinedKey] = new List<GameObject>();
    }
    combinedClusters[combinedKey].Add(agent);
}
return combinedClusters;
```

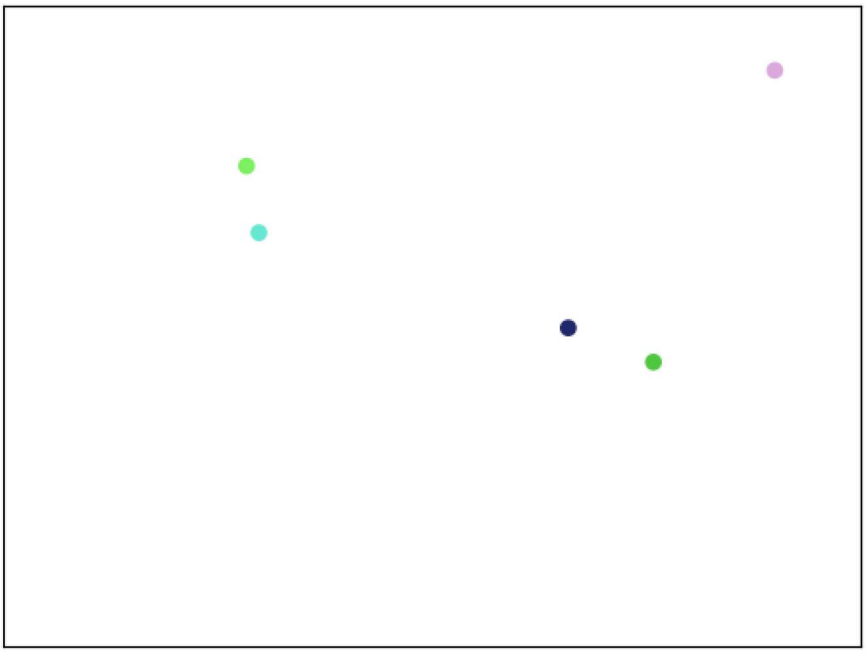
6 | Ažuriranje navigacijskih mreža

Najveći problem naivne implementacije algoritma generiranja dinamičkih mreža kretanja je konstantno ažuriranje mreža kad god se agenti pomaknu. Mreže bi trebalo ponovno ažurirati samo kad je to potrebno, odnosno kad će jedan od agenata uskoro ostati bez prostora za kretanje. Pogledajmo primjer na slici 6.1.



Slika 6.1: Agenti u navigacijskoj mreži imaju dovoljno prostora za kretnju

Vidljivo je da unutar ove navigacijske mreže svi agenti imaju dovoljno prostora za kretanje. Stoga, nema potrebe ažurirati mrežu svaki put kad se jedan od agenata pomakne.

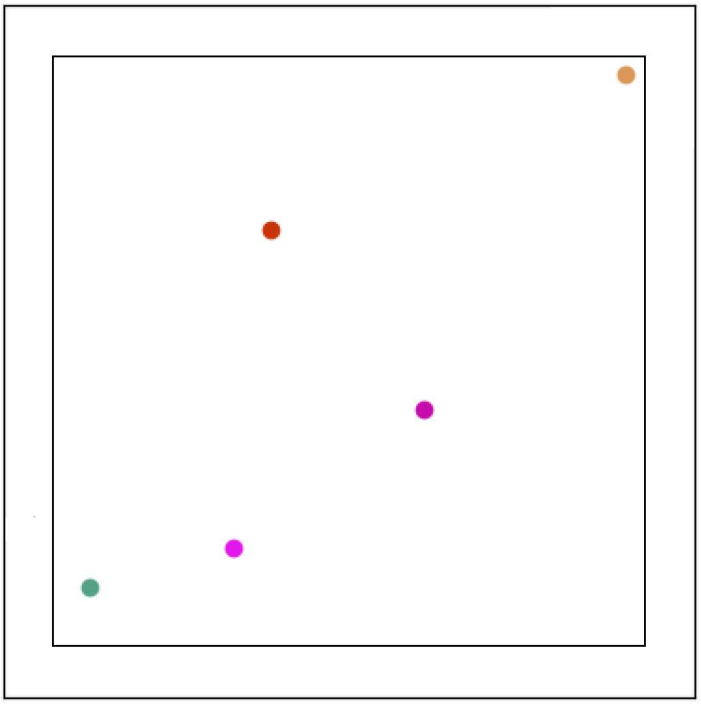


Slika 6.2: Agent u navigacijskoj mreži je blizu ruba mreže

Promotrimo agenta boje lavande na slici 6.2. On se jako približio rubu navigacijske mreže, tako da ćemo istu uskoro morati ažurirati kako agent ne bi ostao bez prostora za kretanje.

6.1 Detekcija agenata koji su blizu ruba navigacijske mreže

Kako bismo odredili treba li se navigacijska mreža ažurirati ili ne, unutar pravokutnika navigacijske mreže stavit ćemo još jedan pravokutnik smanjenih dimenzija, kao na slici 6.3.



Slika 6.3: Navigacijska mreža s manjim pravokutnikom

Mrežu ćemo ažurirati onda i samo onda kada jedan od agenata izađe iz ruba manjeg pravokutnika.
Uočimo da, na slici 6.3, narančasti agent gotovo dodiruje rubove navigacijske mreže. U tom sučaju potrebno je ažurirati navigacijsku mrežu. Provjeru potrebe ažuriranja navigacijske mreže provjeravamo u *Update* metodi klase *DynamicNavMeshController*

```
1  void Update()
2  {
3      // Only update if the navmeshsurface is not in the update
       process already
4      if (State == DynamicNavMeshState.Ready)
5      {
6          foreach (var agent in agentsInside)
7          {
8              if (
9                  !smallerBounds.Contains(agent.transform.position)
10             )
11             {
12                 // Mark for update
13                 GlobalNavMeshController.MarkForUpdate();
14                 foreach(var agentInside in agentsInside)
15                 {
16                     agentInside.GetComponent<NavMeshAgent>().
       enabled = false;
17                 }
18                 break;
19             }
```

```
20              }
21          }
22  }
```

Agente unutar mreže provjeravamo samo ukoliko je mreža izgrađena. Provjera potrebe za ažuriranjem u potpunosti je implementirana prema prethodno objašnjenom algoritmu.

7 | Kretanje agenata

Agenti se u simulaciji mogu kretati na 2 načina, nasumično i ručno. Kod nasumičnog načna kretanja, agenti se kreću prema nekim predefiniranim točkama u prostoru, a točka prema kojoj se agent kreće određena je nasumično. Kod ručnog načina kretanja, svakog agenta je moguće pojedinačno pomicati u prostoru.
Implementacija kretnje agenata sadržana je u dvije klase: *AgentMovement* i *AgentManager*. Klasa *AgentMovement* zadužena je za kontroliranje pokreta jednog agenta, dok je klasa *AgentManager* zadužena za upravljanje odabirom agenata i ciljnih točaka u ručnom načinu kretanja.

7.1 Kretanje agenta do točke

Ukoliko se točka nalazi unutar granica navigacijske mreže agenta, dovoljno je agentu reći da se pomakne do određene točke naredbom

```
1     navMeshAgent.SetDestination(destination);
```

gdje je *destination* instanca klase *Vector3*, a navMeshAgent instanca klase *NavMeshAgent* koja je pridružena agentu kao Unity komponenta.
Ukoliko se pak točka nalazi van navigacijske kreže, agent vrlo često neće znati doći do nje.
Kao što je prethodno navedeno, kretanjem agenta upravljamo u klasi *AgentMovement*. Agentu naređujemo da se pomakne do točke u prostoru koristeći metodu *MoveToPosition*.

```
1 public void MoveToPosition(Vector3 position)
2 {
3     if (navMeshAgent != null)
4     {
5         originalDestination = position;
6         TryMoveToPosition(position);
7     }
8 }
```

Metoda će spremiti ciljnu točku kao krajnju destinaciju kretnje agenta i pozvat će drugu metodu, *TryMoveToPosition*, za samu kretnju agenta.

```
private void TryMoveToPosition(Vector3 position)
{
    // Check if the position is on the NavMesh
    NavMeshHit hit;
    if (NavMesh.SamplePosition(position, out hit, navMeshAgent.
    height, NavMesh.AllAreas))
    {
        // Position is on the NavMesh
        navMeshAgent.SetDestination(hit.position);
    }
    else
    {
        // Move to the nearest valid point
        Vector3 nearestPoint = FindNearestNavMeshPoint(position);
        if (nearestPoint != position)
        {
            navMeshAgent.SetDestination(nearestPoint);
        }
    }
}
```

Metoda prvo provjerava je li ciljna točka unutar navigacijske mreže. Ako je, agent će direktno biti usmjeren prema toj točki. Ako nije, agent će se pomaknuti prema najbližoj točki unutar nekog predefiniranog radijusa. Najbližu određujemo metodom *FindNearestNavMeshPoint*

```
private Vector3 FindNearestNavMeshPoint(Vector3 position)
{
    NavMeshHit hit;

    if (NavMesh.SamplePosition(position, out hit,
    MAX_SEARCH_DISTANCE, NavMesh.AllAreas))
    {
        return hit.position;
    }
    return position;
}
```

U *Update* metodi klasa stalno provjerava je li agent došao do trenutne odredišne točke, te je li ta točka *originalDestination* ili je to trenutno najbliža dostupna točka. Agent će se kretati do trenutno najbliže dostupnih točki dok god ne dođe do krajnjeg odredišta.

```
void Update()
{
 ...
 else if(navMeshAgent.remainingDistance <= navMeshAgent.
    stoppingDistance) {
                if(
                    navMeshAgent.destination.x != ((Vector3)
    originalDestination).x
                    || navMeshAgent.destination.z != ((Vector3)
```

```
    originalDestination).z
8                 ) {
9
10                    TryMoveToPosition((Vector3)originalDestination)
    ;
11                }
12     ...
```

Literatura

[1] Vinković Tomislav, GitHub - DynamicUnityNavmeshGeneration. GitHub. dostupno na https://github.com/TomislavVinkovic/DynamicUnityNavmeshGeneration, 2024.

[2] Unity, Unity - Documentation. dostupno na https://docs.unity3d.com/Manual/index.html, 2022.

[3] Unity Learn, Unity Learn - John Lemon's Haunted Jaunt: 3D Beginner. dostupno na https://learn.unity.com/project/john-lemon-s-haunted-jaunt-3d-beginner, 2021.

[4] Unity Learn, Unity Learn - Unity NavMesh. dostupno na https://learn.unity.com/tutorial/unity-navmesh, 2020.

Sažetak

U ovom radu istražuju se metode optimizacije generiranja dinamičkih mreža kretanja u Unityu. Ujedno se opisuje i izrada aplikacije koristeći Unity 2022.3. programsko okruženje, pomoću kojeg ćemo vizualizirati generiranje mreža kretanja i kretanje agenata po 3D virtualnom okruženju. Implementacija sadrži optimiziran sustav generiranja dinamičkih mreža kretanja, jednostavno 3D okruženje sa preprekama za demonstraciju tehnika optimizacije i skriptu za smisleno kretanje AI agenata. Programski kod projekta napisan je u C# programskom jeziku.

Ključne riječi

dinamičke mreže kretanja, Unity 2022.3., AI agenti, optimizacija, 3D okruženje, izrada aplikacije, C#

Optimization of dynamic pathfinding network generation in Unity

Summary

In this paper, the methods for optimizing the generation of dynamic pathfinding networks in Unity are investigated. Additionally, the development of an application using the Unity 2022.3 software environment is described, which will be used to visualize the generation of pathfinding networks and the movement of agents in a 3D virtual environment. The implementation includes an optimized system for generating dynamic pathfinding networks, a simple 3D environment with obstacles to demonstrate optimization techniques, and a script for meaningful movement of AI agents. Program code is written in C.

Keywords

dynamic pathfinding networks, Unity 2022.3., AI agents, optimization, 3D environment, application development, C#

Životopis

Zovem se Tomislav Vinković i rođen sam 16. siječnja 2003. godine u Osijeku. Od 2009. do 2010. pohađao sam Osnovnu školu Viktor Car Emin u Donjim Andrijevcima. Svoje osnovnoškolsko obrazovanje nastavio sam u Osnovnoj školi dr. Franje Tuđmana u Belom Manastiru, a završio sam ga 2017. godine. Te iste godine upisao sam Prvu srednju školu u Belom Manastiru. Svoje srednjoškolsko obrazovanje završio sam 2021. i stekao zanimanje tehničar za računarstvo. Te iste godine upisao sam Prijediplomski studij Matematika i računarstvo na tadašnjem Odjelu za matematiku u Osijeku, koji sam završio 2024. godine.
U ožujku 2022. godine pohađao sam studentsku praksu u tvrtki Factory. U travnju 2023. zaposlio sam se u tvrtki Customised Cloud u Osijeku kao full-stack developer. Od studenog 2023. do rujna 2024. također sam bio zaposlen u tvrtki Gepek, startupu koji želi omogućiti privatnim osobama prijevoz paketa na ruti koju voze.